A Comparison of Scheduling Algorithms in MapReduce Systems

Karthik Kotian (kkk3860), Jason A Smith (jas7553), Ye Zhang (yxz4728)

1. Overview

MapReduce is a model used for processing large data sets in parallel across a distributed cluster of computers. Generally, the process happens in two phases, a map phase and a reduce phase. In the first phase, a very large problem is split into smaller tasks and then distributed among many peers. Those peers then individually solve the smaller problems. Once the smaller problems have been evaluated, their results are gathered in the second phase and reduced into one final result. Ultimately, harnessing a large number of peers and applying a divide-and-conquer methodology can, dramatically reduce the completion time of jobs, when compared to using a single peer.

The architecture for a MapReduce system can be thought of simply as one front-facing “master node”, that accepts jobs from clients, connected to many back-end “worker nodes”, referred to as a cluster (see the figure below). Jobs are divided by the master node and distributed among the worker nodes.

![MapReduce Architecture Diagram]

One critical piece of any MapReduce system is the scheduler used by the master node. When jobs come in to the master node’s queue they must be divided among worker nodes in some algorithmic fashion. A good scheduler will use the cluster’s resources to its fullest potential, resulting in overall faster job completion time. Maximizing a cluster's potential includes utilizing processing power effectively, minimizing network traffic overhead (data locality), maximizing throughput, minimizing latency, and maintaining fairness.

In this project, we plan to simulate three different scheduling algorithms in MapReduce. We first started our investigation by referencing research articles about existing scheduling algorithms and their respective characteristics. We chose mean completion time as the basis of comparison, and then chose three candidate algorithms to compare in a simulated MapReduce system using that criteria. The schedulers are outlined briefly below.

The First In First Out (FIFO) scheduler selects the first job in the queue, pops it off, and splits the task evenly among all the worker nodes in the cluster. The Fairness scheduler, conversely, splits the resources evenly among all the tasks, i.e., every task gets an equal share of resources. Lastly, the Capacity scheduler is an adaptation of the Fairness scheduler built for large clusters. The Capacity scheduler utilizes sub-queues and priorities within the master node to make decisions as to whom to schedule tasks.

2. Statement of hypothesis

We are testing the following three hypotheses based on their respective algorithm.
FIFO:
Least average execution time of jobs compared to other two algorithms, when the number of tasks is less than the cluster size.

Fairness:
Least average execution time of jobs compared to other two algorithms, when the cluster size is less than the number of tasks.

Capacity:
Least average execution time of jobs compared to other two algorithms, when the cluster size is less than 100.

3. A description of the approach you took to prove or disprove your hypothesis

We have written a discrete event simulation program to simulate the working of MapReduce with each scheduler. For collecting the data during simulation we have kept the cluster size constant and the worker node queue size constant. We will be linearly increasing the number of number of jobs feed to the cluster and collect the response time for the computing all the jobs in the queue. We will perform each test on each of the schedulers and compare them based on average task completion time. We will also test the performance of these algorithms on small and large clusters to test our hypotheses more accurately.

4. Analysis of first research paper [1]

Often times, clusters in a Hadoop system are not homogeneous. Each node may have a different set of strengths and weaknesses, and capabilities to process certain tasks faster and more efficiently than other nodes. Additionally, each job coming into the system often is better suited for a particular set of nodes. Assuming both job and resource homogeneity can lead to inefficiencies in scheduling and ultimately inefficiencies in job completion time.

Novel Contribution
The research paper, An Adaptive Scheduling Algorithm for Dynamic Heterogeneous Hadoop Systems, proposes a new scheduling algorithm, which takes into account heterogeneity of jobs and resources, fairness and dissatisfaction, and minimum share requirements. By classifying jobs and resources into categories, jobs can be more intelligently assigned to resources of the appropriate category for better efficiency. This adaptive scheduler also makes novel contributions such as reducing communication costs by not necessarily maximizing a task’s distribution, reducing search overhead for matching jobs and resources, and increasing locality.

Usefulness
fair-scheduling algorithms. It introduced a lot of background information on the existing schedulers and where their shortcomings lie. We planned on implementing a simulation of FIFO and fair-scheduling, so we used information in this paper to capture those shortcomings. The paper used mean completion time as one of their benchmark
comparisons, which was useful because our hypothesis is based entirely on comparing mean completion time.

5. Analysis second research paper [2]

In data intense computing MapReduce as to process large amount of data. To felicitate that large amount of data is moved across the network. To reduce network traffic and minimize data movement Hadoop uses its default scheduling strategy that takes a task-by-task approach. In this scheduling, when a node reports that it has idle slots, the master scans the queue to find the task that can achieve the best data locality. Master searches for a task whose input data are located on that node. If the search fails, it searches for a task whose input data are located on the same rack as the node. If the search fails again, it randomly assigns a task. But this approach is not optimal.

This paper proposes a new algorithm lsap-sched that yields enhance data locality. In this scheduling algorithm, they mathematically reformulate the Hadoop’s scheduling problem by using a cost matrix to capture the cost of data staging [2]. But pure data locality may result in unfair or limited use of resources. So, they also integrated fairness to data locality and proposed a better algorithm lsap-fair-sched. This algorithm allows the users to adjust the degree of data locality and fairness to get the optimum response time by allowing users to express the tradeoffs easily [2]. The paper mentions their proposed algorithm reduces data locality cost by 70–90 % when compared to Hadoop’s dl-sched scheduler.

Problems Addressed
- Fairness and Data Locality conflict with each other when used together. Strict Fairness degrades data locality. Pure data locality enhances reduces fair use of resources.
- In a shared cluster, there can be a scenario where small number of users overwhelms the resources of whole cluster. The proposed scheduler lsap-fair-sched avoids this scenario.

Novel Contribution
- The paper investigates the tradeoff between data locality and fairness cost. The paper explains, why strict use of either schedulers reduces optimum resource utilization and eventually the response time to execute a job. The idea of using data locality and fairness together intrigued us.
- Proposes algorithm lsap-sched gives optimal data locality by utilizing the well-known problem LSAP. In addition, it integrates fairness with lsap-sched scheduler. This scheduler is called as lsap-fair-sched.
- The following is the brief explanation of the lsap-fair-sched scheduler.

Input: α - Data Locality Cost Matrix DLC scaling factor for non-data local tasks. If α is large, DLC stands out and the scheduler favors data locality.
β - FC scaling factor for tasks that are beyond its group allocation

Output: is a matrix that gives the assignment of tasks to idle map slots.

Important Functions:
1. First find the set of nodes with idle slots.
2. Find the set of tasks whose input data are stored on nodes with idle slots.
3. Calculate slot to occupy (sto) matrix of all groups.
4. Calculate task Fairness Cost matrix (FC).
5. Calculate Data Locality Cost matrix (DLC).
6. Calculate the final cost matrix by adding FC and DLC.
   \[ C = FC + DLC. \]
7. If C is not a square matrix then expand it to one.
8. Finally, \textit{lsap} algorithm is used to find the optimal assignment, which is subsequently filtered and returned.

Usefulness
During the research for our team project we found the data locality is an important aspect in scheduling. Hence chose this paper to understand and investigate data locality scheduler in detail. Although we did not implement \textit{lsap-fair-sched} for our simulation because of its complexity, its comparison with Fairness and Hadoop's Data Locality Scheduler gave us valuable insights about Fairness Scheduler (which we have used for our simulation).

6. Analysis of third research paper [3]

The third paper is: Improving MapReduce Performance in Heterogeneous Environments. In practice, the homogeneity assumptions do not always hold. Hadoop’s scheduler can cause severe performance degradation in heterogeneous environments. Thus, in this paper, the author introduced why some scheduler algorithm may fail under heterogeneous environments. Meanwhile, the paper also discussed a new scheduler algorithm called LATE. To ensure the validity of its statement, the author also implemented a series experiment to compare LATE with current algorithms.

Problems Addressed
As mentioned before, there are a few assumptions in Hadoop’s scheduler:
1. Nodes can perform work at roughly the same rate.
2. Tasks progress at a constant rate throughout time.
3. There is no cost to launching a speculative task on a node that would otherwise have an idle slot.
4. A task’s progress score is representative of fraction of its total work that it has done. Specifically, in a reduce task, the copy, sort and reduce phases each take about 1/3 of the total time.
5. Tasks tend to finish in waves, so a task with a low progress score is likely a straggler.
6. Tasks in the same category map or reduce, require roughly the same amount of work. In practice, all these assumptions may have problems. For the first two assumptions, they may fail, as nodes can be slow for other reasons. For others, they may fail tasks from different generations will be running concurrently, even in a homogeneous environment with long enough job. Thus, we need new algorithm that could fixed all these problems, and make the most correct decision about how to assign task to nodes.

Novel Contribution
The novel contribution of this paper is a new thought and the author established new algorithm. The strategy of it is always speculatively execute the task that we think will finish farthest into the future. Previously, people use the fraction of input data read to be the progress score. For example, the execution is divided into three phases, each of which accounts for ⅓ of the score. They are copy, sort and reduce. But, this is not fair in most cases. The copy
phase of reduce tasks is the slowest, because it involves all-pairs communication over the network. Thus, if compare one node almost finish copy phase and the other node just finish copy phase, it is hard to say their efficiency is different significantly. Currently, the author use progress rate to decide which one is slower. It equals to progress score after divide T, which is the amount of time the task has been running for. It has a series of advantages: 1. It is robust to node heterogeneity, because it will re-launch only the slowest tasks, and only a small number of tasks. 2. LATE takes into account node heterogeneity when deciding where to run speculative tasks. 3. By focusing on estimated time left rather than progress rate, LATE speculatively executes only tasks that will improve job response time, rather than any slow tasks. According to the result analysis, it is also apparent that the performance of LATE is much better than previous algorithm in various environments.

Usefulness
In reality, most assumptions that we mentioned previously could not be ensured all the time. Thus, it is worth to discuss all these cases. This paper helped us understand what issues we may face when we implement the scheduler we selected before.

For the reason of complexity and lack of data, we did not implement it, but it offers a good reference for us to evaluate the scheduler that we will implement in our project.

7. Software Design

Figure 2: Flow Diagram of the project

Classes used from Parallel Java Library:
Class Event [5]
It is the abstract base class for an event in a discrete event simulation [5].
Class Simulation[5]
Class Simulation provides a discrete event simulation [5].

Class MasterNode:
This class contains the main method and runs the discrete event simulation. In figure 2 we can see that, MasterNode initializes Job objects and WorkerNode objects (Cluster) and passes these objects to Scheduler Object. It creates the Class Simulation object and executes Simulation's run() method to start the simulation. This class also gives the Response Time of the scheduler, mean response time of WorkerNode and Standard Deviation. It also generates plot of response time vs. number of jobs.

- **Class TestMasterNode:**
  This class is similar to MasterNode class. The only difference is it runs all the three scheduler simultaneously and plots the graph of all the three schedulers response time w.r.t. Number of jobs. Also it generates no data to save heap memory.

Class WorkerNode:
The Worker Node calculates the response time for every Task it gets executed. Every WorkerNode cumulates the response time of all the Task it executes in a Simulation. The WorkerNode with the highest cumulative response time gives the response time for that simulation. The WorkerNode also maintains queue for Task's. For this simulation we maintain a constant queue size of 5.

Class Scheduler:
This is a abstract class. It maintains the Simulation class object. It also collects the response time of each task. The classes extending Scheduler implements generateRequest() method. generateRequest() creates one or more Events and adds them to simulation by calling doAfter() method. generateRequest() calls each Event's perform() method, in order according to the events' simulation times, as returned by each event's time() method[5]. This method is called recursively till there is no events (Task objects) in the queue.
The following classes extend Scheduler.

- **Class FifoScheduler**
  This class implements the FIFO scheduler algorithm. The scheduler breaks down Job object to Task objects. These Task objects are then distributed to the all the
WorkerNode’s queue that is not full. A Task is feed to the WorkerNode’s queue with least or no Task in it.

• **Class FairScheduler**
  This class implements the Fair scheduler algorithm. The scheduler breaks down Job object to N number of Task objects where N is the cluster size.

• **Class CapacityScheduler**
  This class implements the Capacity scheduler algorithm. The scheduler breaks down Job object to Task objects. The number of Task object depends on the size of the inner cluster. WorkerNode objects with same id are part of the same inner cluster and consumes Task object with same Task Id as theirs. If the queue of WorkerNode’s is full in the same inner cluster then it allocates the Task to WorkerNode with least or no Task in it. The priority of the task is given by its id.

**Class Job:**
This class maintains the Job id and complexity. Complexity of the job ranges from 1 to 10 where Job with complexity 1 will require the least amount to execute and job with complexity 10 will require 10 times more time to execute, when compared to job with complexity 1.

**Class Task:**
This class calculates the response time for each Task to get executed. The response time is calculated using the Job complexity. If the complexity of the job is 1 then it takes one unit of time, if the complexity is n the it takes $n \times \text{complexity}$ unit of time. It also maintains the response time in ListSeries.

**8. Developer's Manual**

To run the project on RIT Computer Science server.
- Extract the files from the DistributedMind.jar file
- Login to any RIT Computer Science server using the following command
  ```
  ssh -X user_name@server_name.cs.rit.edu.
  ```
  For example, `ssh -X kkk3860@idaho.cs.rit.edu`
- Copy all the `.java` files to a same folder. You can create a folder by using `mkdir folder_name`.
- The code uses Parallel Java (PJ) Library. When compiling and running PJ programs, you must use JDK 1.5. PJ uses features of the Java language and platform introduced in JDK 1.5 and will not compile with earlier JDK versions [5].
- [5] Compiling and executing Java programs that use PJ, if you have installed the executable distribution, you must set your classpath to include the PJ JAR file.
  Here is an example of a command for the `bash` shell to set the classpath to the current directory plus the PJ JAR file:
  ```
  export CLASSPATH=./home/fac/ark/public_html/pj.jar
  ```
  Here is an example of a command for the `csh` shell to set the classpath to the current directory plus the PJ JAR file:
  ```
  setenv CLASSPATH ./home/fac/ark/public_html/pj.jar
  ```
  - Compile the * .java files by executing the following command.
To run the project on Eclipse IDE on your personal system

- Start the Eclipse IDE.
- Create a Java Project.
  File > New > Java Project.
  Name your project and press Finish
- Now we need to add the Parallel Java Library to our project.
  First download the executable or source distribution of Parallel Java Library. The link for the website is here: http://www.cs.rit.edu/~ark/pj.shtml
- Change the file name to pj.jar.
- Right click on the project name in Project Manager of Eclipse
  Go to Properties. Select Java Build Path and then click on Libraries tab.
  Select Add External JARs..... and navigate to pj.jar. Select pj.jar and press Open. The Parallel Java Library is added to the Java Project.
- Extract the files from the DistributedMind.jar file. Copy all the .java to your Java project.
- In the tool bar Select Project and select Build Automatically.

9. User's manual for our software

Class MasterNode contains the main method. The Usage for the class is
Usage: java MasterNode <sched> <J> <T> <N> <seed>
where
<sched> = 1. FIFO 2. Fairness 3. Capacity
<J> = Maximum number of jobs
<T> = Number of Trails
<N> = size of the cluster
<seed> = Random seed
The code will output the response time, mean response time of each worker and
standard deviation for Jobs from 1 to maximum number of Jobs given in the usage. It will
also generate a plot of response time vs. number of jobs.

To run the project on RIT Computer Science server
- Compile the software as mentioned in the above section.
- Run the command mentioned in the Usage.
- The code will give the data and generate a plot of that data.

To run the project on Eclipse IDE on your personal system
- Right click on the MasterNode class in Project Manager.
- Select Run As > Run Configurations....
- Select Argument tab and enter the arguments in the Program arguments text box
  and press Run.
- This should generate the output data in the Console and generate a plot in X11
  of the output data.

10. Data Collected

To test our hypothesis we have simulated our code with 9 test cases. All the generated
data and parameters of the test cases are given in the folder GeneratedData in our
DistributedMind.jar file. Since the amount of data is large, in this section we have
mentioned only the plots generated from the test case data.
Figure 4: Test case 2 - <Cluster size: 100> <Maximum number of jobs: 30> <Trails: 100>

Figure 5: Test case 4 - <Cluster size: 100> <Maximum number of jobs: 80> <Trails: 100>
11. Analysis of our data
Let us analyze from the above given plots if our hypothesis is approves or disproves the data generated:

**Hypothesis 1:** FIFO’s Least average execution time of jobs, compared to other two algorithms, when # of tasks < cluster size.

From fig. 4 and 5 we can say that FIFO’s response time is better that Capacity and Fair Scheduler. The data also satisfies the condition of number of task to be less than cluster size. For Test case 2 and 4 the cluster size is 100 and maximum number of jobs are 30 and 80 respectively. Therefore the original hypothesis is true.

**Hypothesis 2:** Fairness’s Least average execution time of jobs, compared to other two algorithms, when cluster size < # of tasks.

The data generated form Test case 6 shows that Fairness Scheduler as the best response time. In fig. 6, for cluster size =10 and maximum number of job = 200, Fairness scheduler out performs FIFO and Capacity scheduler. But as we increase the cluster size (keeping the number of jobs more than cluster size) the performance of Fairness scheduler decreases. In test case 6, FIFO performs better than Fairness. Also performance of Fairness falls behind Capacity scheduler when numbers of jobs are more than 140. This can be seen in figure 7.

From the data we can say that Fairness performs better than other two scheduler when the cluster size is very small. But as the cluster size increases, the other schedulers outperform Fairness. Hence we can say that the original hypothesis is false.

**Hypothesis 3:** Capacity’s Least average execution time of jobs, compared to other two algorithms, when cluster size > 100.

From the collected data, we observed that Capacity performs poorly when cluster size is less than 30. The performance of Capacity increases very sharply when the cluster size is more than 30. From test 2 and 4 we can see that FIFO performs better than Capacity Scheduler. That is due the less number of jobs compared to cluster size. In test case 7 we can see that performance of capacity scheduler gets better as the number of tasks increases. In fig. 7, Capacity’s response time performs better than fairness scheduler after 140 jobs. In test case 7, Capacity will eventually outperform FIFO when the numbers of jobs increases. We could not test out code for jobs more than 200 because of Java heap overflow.

From the generated data we can say that Capacity will perform better than the other two on a large cluster only when the number of jobs in the queue is very large. Therefore the original hypothesis is false.

12. **A discussion of possible future work**

- We can simplify the design of the code by removing the Task class. This reduces the heap usage and allows us to test the code with more number of iterations. This will increase uniformity in the collected data.
- We would like to introduce data locality in this simulation. Data locality is a very important were data intensive computation is required.
- We would also like to implement *lsap-fair-sched* scheduler introduced in Research paper 2 [2]. This algorithm introduces data locality and integrates it with Fairness. It will be interesting to see the simulation of this algorithm.
13. A discussion of what you learned from the project

The project helped us understand how simulation can be used to implement a discrete event simulation and test those implementation on various testing factors. Simulation requires us to write code and don’t require any physical implementation of the testing environment, which makes testing simple and cost effective. This project helped us understand MapReduce and it’s working in detail. We also learned how Scheduler’s works in MapReduce framework and use of appropriate scheduler can reduce the response time to execute a job. From our research paper investigation we learned that data locality is an important aspect in a data intensive computation in MapReduce. Proper implementation and use of data locality can reduce a network traffic, decrease latency and enhance the response time of the computation.

Simulation can be used for linear regression and testing our hypothesis in java. We found use of Parallel Java Library to be simpler and flexible than Matlab and R.

14. A statement of what each team member did on the project

Ye Zhang
- Introduced research paper *Improving MapReduce Performance in Heterogeneous Environments* [3] to the team.
- Implemented the FIFO Scheduler class and test it.
- Implemented the TestMapReduce class, which simulates all the three algorithm simultaneously.
- Tested the code with the team and collected the data for analysis.

Jason Smith
- Introduced research paper *An Adaptive Scheduling Algorithm for Dynamic Heterogeneous Hadoop Systems* [1] to the team.
- Implemented the initial design of the project along with Karthik.
- Implemented the FairScheduler class and test it.
- Tested the code with the team and collected the data for analysis.

Karthik Kotian
- Introduced research paper *Investigation of Data Locality and Fairness in MapReduce* [2] to the team.
- Implemented the initial design of the project along with Jason.
- Implemented the CapacityScheduler class and test it.
- Tested the code with the team and collected the data for analysis.
15. **List of references**


[2] Investigation of Data Locality and Fairness in MapReduce., Zhenhua Guo, Geoffrey Fox, Mo Zhou, Indiana University, Bloomington, IL, MapReduce 2012 Proceeding of third international workshop on MapReduce and its application date.

